Parallel Particle Simulations using the POOMA Framework[*]

Julian C. Cummings[†] William F. Humphrey[†]

Abstract

The POOMA Framework is an object-oriented library and application suite implemented in C++ which enables the parallel simulation of the dynamics of interacting particles and fields. Using templates, POOMA provides a flexible and intuitive syntax to express global field and particle operations and creates applications with near hand-coded performance. POOMA is employed at present in multi-material hydrodynamics calculations, gyrokinetic plasma dynamics, and Monte Carlo neutron transport simulations. Currently under development within the POOMA Framework are interactive visualization tools for use in all applications developed using POOMA.

1 Introduction

The POOMA (Parallel Object-Oriented Methods and Appliations) Framework is a library of software components for use in the development of scientific computational applications employing parallel computers. It provides a set of flexible software abstractions of physical quantities such as particles and fields, and an efficient, data-parallel syntax for expressing the numerical solutions to the equations of motion or state governing a physical system [10, 9].

In many simulations of complex physical problems, a recurring set of methods and software abstractions for the physical domain and numerical modeling procedures often arise. These recurring concepts include: data structures representing particles with associated position, velocity, and other attributes; data structures representing field quantities discretized on a mesh over some spatial domain; and operations involving these particle and field quantities. Many simulations also require large memory and CPU resources to study phenomena which occur across large domains in time or space. The purpose of the POOMA Framework is to help decrease the time for both the development and the execution of scientific computional applications by recognizing these two features of such applications and applying the techniques of parallel computation and object-oriented design. The POOMA Framework includes objects for commonly occuring physical quantities such as particles and fields, and encapsulates the use of parallel algorithms within the object-oriented structure of the Framework.

A number of packages have been developed to aid in the implementation and execution of scientific computing applications, such as the extensive Parallel ELLPACK package [6]. An important feature of the POOMA Framework is its support for both particle and field quantities in a single integrated, extensible toolkit, which allows POOMA to address a wider range of problems than systems which focus primarily on, e.g., solution of systems of partial differential equations. In this paper we focus on the particle simulation features of

[*]This work was funded by the US Department of Energy.

[†]Advanced Computing Laboratory, Los Alamos National Laboratory, Los Alamos, NM.

the POOMA Framework, starting from a discussion of the problem domain addressed by the design of POOMA, and followed by a description of the object-oriented design of the POOMA particle components. We conclude with a discussion of a sample application using the POOMA Framework, a Monte Carlo based neutron transport code which highlights the use of interacting particle and field objects in a single program and demonstrates the use of POOMA's parallel, object-oriented design to express the solution to a physical problem.

2 Problem Domain

POOMA objects and algorithms are designed for use in solving systems of PDE's and ODE's, for a wide variety of initial and boundary conditions, in an arbitrary number of spatial dimensions. For these problems, POOMA provides two basic abstractions of physical quantities, implemented as C++ objects: Fields and Particles. POOMA provides both the data structures (objects) to represent these concepts and methods for calculating the interactions between Particles and Fields. These objects are extensible to new or more complex data structures and algorithms.

2.1 POOMA Fields

A Field object represents a discretization of a continuous field quantity over an underlying mesh. Using the template mechanism of C++, the type of data stored by a Field object may be arbitrary, as may be the number of dimensions D for the Field. Internally, Field data is stored essentially as a D-dimensional array, over an index space selected using POOMA Index objects. The underlying mesh for a Field object may also be specified as, e.g., cartesian, cylindrical, or spherical, and the Field may be cell-centered or vertex-centered. A variety of boundary conditions may be specified for the Field, and separate boundary conditions may selected for each face of the D-dimensional domain.

When running on a parallel architecture, POOMA will automatically partition the data in a Field among processor nodes into local Field subdomains, and the user may select which axes of the Field to parallelize. POOMA also maintains, when running in parallel, *guard* or *ghost* cells along local Field boundaries which store copies of data from the edges of neighboring local Fields, a common optimization for computation of stencil operations in PDE solvers.

2.2 POOMA Particles

Particle container objects in the POOMA Framework store a collection of individual particle elements, where each single particle has some associated number of attributes such as position, mass, velocity, etc. Typical examples of Particle objects are representations of atoms in a molecule or crystal, or representations of electrons or ions in a plasma. As for Fields, the type and dimension of the attribute data for each particle may be selected using the template feature of C++. Particles may be dynamically created and destroyed during a simulation.

While Field quantities are used quite frequently (but not exclusively) for the solution of PDE's, the range of problems for which particles may be required is quite diverse. Particle simulations employ many different techniques such as molecular dynamic, Monte Carlo, Langevin dynamic, or particle-in-cell (PIC) algorithms. These algorithms assume quite different interaction potentials between particles and possibly with external Fields, and have different requirements for efficient parallelization. To account for this, POOMA provides several unique mechanisms for assigning particles to nodes when running in parallel, and the user is free to choose the optimal partitioning strategy based on the particular type of simulation being performed. The Particle objects in POOMA automatically handle reassignment of particles to processors when necessary.

A common requirement of particle-based simulations is the ability to compute the interactions between a particle and its nearest neighbors. Particle container objects in POOMA will locate those particles within a selected interaction region of a particle, for use in, e.g., molecular dynamics calculations of the Coulombic interaction potential between atoms. A constant interaction radius may be used for all the particles, or the interaction radius may be specified for each particle separately.

2.3 Particle-Field Interactions

POOMA applications may optionally employ just Field objects, just Particle objects, or both, which is an important feature of the POOMA Framework. POOMA provides the functionality to compute interactions between Particles and Fields which arise in algorithms such as, but not limited to:

- the particle-in-cell (PIC) method [4] which solves the Poisson equation $\nabla^2\phi = -4\pi\rho$ to find the electrostatic potential field ϕ due to a charge distribution ρ resulting from a charged particle population;

- the particle-particle-particle-mesh (PPPM) method [5], a combination of the molecular dynamics technique for calculating the close-range electrostatic interactions between charged particles, and the PIC method for computing the long-range electrostatic forces.

POOMA provides methods for scattering particle attributes (such as charge) onto an underlying Field (such as charge density ρ), and for gathering a Field quantity such as the electric field $\mathbf{E}$ to a particle position $\mathbf{r}$, using one of a number of different interpolation methods. These abilities require, when working in parallel, the use of a Particle distribution mechanism which maintains particles local to the processor which owns the associated local Field subdomain. The object-oriented nature of POOMA simplifies the task of matching the Particle layout mechanism and Particle object behavior to the requirements of the simulation.

3 Object-Oriented Design

The POOMA Framework is implemented as a library of C++ classes, making extensive use of the template facilities of C++. Templated classes allow one to implement general objects such as Particle or Field containers for arbitrary data types. The use of templated classes is one mechanism for software reuse in POOMA, an important benefit of using an object-oriented language such as C++ for scientific computing.

The design of the POOMA Framework makes use of several features of objected-oriented design, including:

- Data encapsulation, which associates data structures and the methods which operate on this data as a single entity, with a common user interface;

- Inheritance, a feature which allows one to derive new objects from existing ones, adding new functionality without the need to reimplement the previous code;

- Polymorphism, the ability to define sets of distinct objects with different functionality but a common interface, which makes it possible to support wide ranges of behavior with the same code, contributing to software reuse.

The Particle and Field objects in POOMA use data encapsulation and inheritance; polymorphism is employed in POOMA to provide an abstraction layer which supports several different message-passing libraries such as MPI [3] or PVM [2]. POOMA objects also make significant use of the Standard Template Library (STL) [8], and many of the idioms from this library such as iterators, containers, and algorithms are employed in the design of the Framework components.

3.1 Particle Class Design

The Particle objects in the POOMA Framework act as containers which store the characteristic data for N individual particulate elements. Each individual particle has several attributes, such as position, mass, velocity, etc. The storage, manipulation, and parallel distribution of the particles is achieved using three general categories of Particle objects:

1. *Particle attribute* objects, which store the values of a single attribute for all the individual particles. Particle attribute objects are templated on the type of the attribute data, and act very much like an STL `vector` container.

2. *Particle layout* objects, which perform the function of distributing individual particles from a particle container object among the nodes of a parallel machine. POOMA provides several different particle layout objects, tailored to the needs of different particle interaction mechanisms. For example, a particular layout object will distribute particles based on their position relative to a given Field object, in order to maintain locality between particles and nearby Field grid points.

3. *Particle container* objects, which contain a set of particle attribute objects. POOMA provides a standard *base* particle container class from which the user can define their own particle container object, with particle attributes of their choice. This container object is templated on the type of the desired particle layout mechanism.

The general method for using the POOMA Particle objects is to (1) select a particle layout object based on the particle interaction mechanism, (2) define a new user-customized particle container object, derived from a Particle base class provided by POOMA, and (3) create (instantiate) copies of this new particle container object, specifying the desired layout mechanism and initializing new individual particles within the container. By separating the parallel layout functionality from the particle container objects, greater flexibility is available to the user to match parallelization strategies to algorithm requirements without the need to reimplement the core particle container functionality. This is another example of software reuse within POOMA.

3.2 Particle Expressions

Using the technique of *expression templates* [11], particle attribute objects may be combined in complex expressions, which are then evaluated for all individual particles local to each parallel node. For example, given a particle container object `P` with attributes `pos` and `vel`, the following statement would update the value of `pos` for all the individual particles:

```
P.pos = P.pos + P.vel * dt;
```

The use of expression templates is a powerful technique which uses the template capability of C++ to evaluate the structure of an expression at compile-time and generate a single, inlined loop without the need for temporary results storage. The POOMA Framework makes extensive use of this technique for Particle and Field objects [9].

3.3 Particle Visualization Capabilities

The ability to interactively visualize the POOMA components in an application is currently being added to the POOMA Framework. Particle objects, as well as Field objects, are being enhanced to provide a simple interface to the user to register objects for visualization and to indicate when data may be safely exported to a visualization component or external application. This capability is encapsulated in a set of POOMA classes, a design with the flexibility to interface with a number of different visualization clients, or, more generally, with many different problem-solving environments.

4 Sample Particle Applications

The POOMA Framework is currently being used in the development of several different scientific application codes that rely upon particle-based algorithms to solve specific problems. Along with the creation of codes that can produce useful new scientific results, the intent of these projects is to explore the benefits of object-oriented design in scientific application code development and to benchmark the performance of POOMA-based applications against comparable existing codes.

4.1 A Simple Example

The flexibility and utility of the POOMA Particle classes is most easily illustrated with a simple example application code. The following short program performs a simulation of charged particles moving in a static electric field using the particle-in-cell method. The code defines a customized Particle class to describe the charged particles, initializes an electric field and a population of these particles, and employs a simple leapfrog scheme to integrate the particles' equation of motion.

```
#include "Pooma.h"                // include Pooma header files
template <class PL> class ChargedParticles : public ParticleBase<PL> {
  public:
  ParticleAttrib<double>  qm;     // charge-to-mass ratio
  PL::ParticlePos_t       V, E;   // velocity and local E field value
  ChargedParticles(PL& pl) : ParticleBase<PL>(&pl) {   // constructor
    addAttribute(qm); addAttribute(V); addAttribute(E);
  }
};
const unsigned int Dim = 2;       // dimensionality of simulation domain
const double pi=acos(-1.0), qmmax=1.0, dt=1.0, E0=0.01;
const int nx=200, ny=200, totalP=4000, nt=50;
typedef ParticleSpatialLayout<double,Dim> playout_t;

int main(int argc, char *argv[]) {
  Pooma pooma(argc, argv);
17   int numnodes=Pooma::getNodes(), mynode=Pooma::myNode();
18   Index I(nx), J(ny);                                 // Indexes for domain
19   FieldLayout<Dim> FL(I,J);                           // FieldLayout
20   playout_t PSL(&FL);                                 // ParticleLayout
21   Field<Vektor<double,Dim>,Dim> EFD(FL);              // create static E field
22   EFD[I][J](0) = -2.0*pi*E0*cos(2.0*pi*(I+0.5)/nx)*cos(4.0*pi*(J+0.5)/ny);
23   EFD[I][J](1) =  4.0*pi*E0*sin(2.0*pi*(I+0.5)/nx)*sin(4.0*pi*(J+0.5)/ny);
24   ChargedParticles<playout_t> P(PSL);                 // Particles object
25   int len = totalP / numnodes;
26   if ((totalP-len*numnodes)>0 && mynode==0) len += totalP-len*numnodes;
27   P.create(len);                                      // add particles
28   Vektor<unsigned,Dim> base(2,3);
29   Vektor<double,Dim> upper(nx,ny);
30   assign_bit_reverse(P.R, P.ID, base, upper);         // initial positions
31   assign_random(P.qm, qmmax, -qmmax);                 // set q/m values
32   P.update();                                         // update particles
33   IntNGP<double,Vektor<double,Dim>,Dim> myinterp; // interpolater
34   for (int it=0; it<nt; it++) {       // main timestep loop
35     P.R = P.R + dt * P.V;             // advance particle positions
36     P.R(0) = fmod(P.R(0)+nx,nx);      // periodic boundary conditions
37     P.R(1) = fmod(P.R(1)+ny,ny);
38     P.update();                       // redistribute particles
39     gather(P.E, EFD, P.R, myinterp);  // get E field values
40     P.V = P.V + dt * P.qm * P.E;      // advance particle velocities
41   }
42   return 0;
43 }
```

The ParticleBase class provides a standard description of a particle population in which each particle has a position `R` (represented as a POOMA Vektor object, a data vector of arbitrary length and element type) and a unique global identification number `ID`. Lines 2–9 define a customized class derived from ParticleBase for this application. After creating some constants in lines 10–12, we begin the main program by creating a Pooma object on line 16, which will manage several run-time tasks and provide useful system information. Next, we must give domain and layout information for the Particle and Field objects we will be using. Line 18 creates Index objects that describe our simulation domain. From these, we can create a FieldLayout object that will manage the decomposition of Fields across processors. In turn, we then generate a ParticleSpatialLayout object that will distribute particles so that their data is on the same processor as Field data near their position, since this is most optimal for performing particle-field interpolation.

With this setup complete, we create and initialize a Field of Vektors to represent the static electric field in lines 21–23. We then instantiate our ChargedParticles class on line 24, and fill it with new particles on line 27 using the `create` member function. Particle attributes can be initialized in various ways, including the use of standard methods such as bit-reversal (as shown in lines 28–30 to set particle positions) and random number generation (as on line 31). The call of `update` on line 32 tells the Particle object to update the parallel distribution of particle data based on the layout and to account for any newly created or destroyed particles. Finally, on line 33, we instantiate an interpolater object, which performs nearest-grid-point interpolation. The Framework provides a standard set of classes that perform well-known interpolation methods, and allows users to add their own interpolation classes. With the Field, Particle and interpolate objects ready, we now perform the main loop to integrate the particle motions in time, as shown in lines 34–41. Note the simple data-parallel syntax used in writing the equations of motion and the straightforward interface for requesting particle-field interpolations. In addition to providing a very flexible description for a particle population, POOMA allows the use of clear, high-level syntax for specifying the scientific phenomena being modeled and manages the issues of on-node performance and parallel computing transparently.

4.2 MC++: Monte Carlo Neutron Transport Simulation

MC++ is an implicit multi-group Monte Carlo neutron transport code written in C++ and based upon an "alpha" version of the POOMA Framework that does not employ advanced C++ template features. It determines the criticality of a system containing fissionable material by finding an asymptotic solution to the transport equations [7]. MC++ was developed within a period of five months using the Particle and Field classes and the POOMA parallel abstractions, and it has been run on a variety of MPP, SMP, and workstation platforms. This simple portability has allowed MC++ to be developed in the relatively robust computing environment of desktop workstations and then easily ported to leading-edge machines such as the Intel Teraflop at Sandia National Laboratory.

As a check of code performance, MC++ has been benchmarked against MCNP [1], a powerful, well-known Monte Carlo transport code written in Fortran 77 and using the PVM library [2] for message passing, on a variety of platforms. Since MC++ calculates on a 3D Eulerian mesh, while MCNP does not employ a mesh at all, a simple test problem was devised in which the system contained a single medium with no boundaries engineered to give a known physics result. Selected timing results of this comparison study are shown in Table 1 for runs of 10 cycles using 40,000 particles.

Platform	Nodes	MCNP (sec.)	MC++ (sec.)
SGI R4k	1	293.9	207.3
SGI R8k	1	115.5	100.9
SGI R10k	1	34.6	50.9
IBM RS6k	1	53.3	119.6
Sun Sparc10	1	105.0	193.1
Cray T3D	1	157.8	225.0
Cray T3D	2	154.8	106.8
Cray T3D	4	219.0	82.2
Cray T3D	8	82.2	46.0

Table 1
Timing comparison of MCNP and MC++ on various platforms.

The single-node runs indicate roughly comparable performance for the two codes, depending upon the architecture used. Parallel runs on the Cray T3D highlight how the design of MC++ around POOMA parallel abstractions allows the code to make better use of massively parallel platforms. It is worth noting that this data was produced with a preliminary version of MC++ that was not tuned in any way for performance. Significant performance gains are expected as the code is refined and migrated to the enhanced version of the Framework described here. Nonetheless, we see that with minimal or no sacrifice in on-node performance, we can produce codes that employ an easily understood, high-level syntax, are straightforward to extend with new modeling capabilities, are quickly ported to new computing platforms, and make efficient use of parallel processing capability.

5 Current Usage

The POOMA Framework is currently employed in a number of applications in collaboration with several research groups at Los Alamos National Laboratory. The Field abstractions of POOMA are used in development of codes which model multi-material hydrodynamics and ocean dynamics. The Particle objects in POOMA are currently in use in a smooth-particle hydrodynamics application, while both the MC++ neutron transport code described in Section 4.2 and a PIC gyrokinetic plasma application employ both Particles and Fields.

POOMA Framework applications have been successfully compiled on a number of workstation and supercomputer architectures, including SGI, DEC, IBM, and Sun workstations as well as the Cray T3D and the Intel Teraflop. Further information about the POOMA Framework may be found on the POOMA web page, `http://www.acl.lanl.gov/Pooma/`, including sample programs and documentation.

6 Acknowledgements

POOMA is developed in the Advanced Computing Laboratory at Los Alamos National Laboratory. The POOMA development team includes Pete Beckman, Paul Hinker, Steve Karmesin, MaryDell Tholburn, and Timothy Williams, and is led by John Reynders.

References

[1] J. F. Briesmeister, ed., *MCNP – A general Monte Carlo N-particle transport code, version 4A*, Publication LA-12625-M, Los Alamos National Laboratory, 1993.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine*, The MIT Press, 1994.

[3] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI*, The MIT Press, 1996.

[4] F. H. Harlow, *The particle-in-cell computing method in fluid dynamics*, Methods Comput. Phys., 3 (1964), pp. 319–343.

[5] R. W. Hockney, S. P. Goel, and J. W. Eastwood, *A 10000 particle molecular dynamics model with long-range forces*, Chem. Phys. Lett., 21 (1973), pp. 589–591.

[6] E. N. Houstis, J. R. Rice, and T. S. Papatheodorou, *Parallel (parallel-to) ELLPACK : An expert system for parallel processing of partial-differential equations*, Math. Comput. Simulation, 31 (1989), pp. 497–507.

[7] S. R. Lee, J. C. Cummings, and S. D. Nolen, *MC++: Parallel, portable, Monte Carlo neutron transport in C++*, in Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 14-17, 1997.

[8] D. R. Musser and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Addison-Wesley, 1996.

[9] J. V. W. Reynders, *The pooma framework: A templated class library for parallel scientific computing*, in Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 14-17, 1997.

[10] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. Tholburn, *Pooma*, in Parallel Programming Using C++, G. V. Wilson and P. Lu, eds., The MIT Press, 1996, ch. 14, pp. 547–587.

[11] T. Veldhuizen, *Expression templates*, C++ Report, 7 (1995), pp. 26–31.